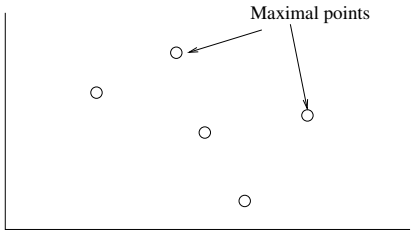
COL 752 Geometric Algorithms

Midterm Sem I 2016 -17 Max 60, Time 2 hrs

1. Given a set S of n points on a plane, a point (x_i, y_i) is *maximal* if $\forall j \neq i$ either $x_i > x_j$ or $y_i > y_j$ (or both). **(7+8)**



(i) Design an $O(n \log n)$ algorithm for finding all the maximal points of S.
The most important observation is that the maximal points form a falling staircase in the first quadrant (as we move in increasing directino of X). These points may not be on the convex hull and this is one of the most common fallacies.

There are various approaches to an $O(n \log n)$ algorithm. Here are a some
Divide and conquer Partition the point set S into roughly equal parts according to x coordinates. Compute the maximal for both halves recursively. To merge, notice that the right staircase R continues to be maximal but the left staircase L may be dominated by points on the right staircase - more specifically, the point with the largest Y coordinate in R will cause all points in L with smaller y coordinates in L **not to be maximal**. This can be clearly identified in $O(n)$ time. So the total running time can be expressed as
$T(n) = 2T(n/2) + O(n)$ implying that $T(n) = O(n \log n)$.

Sweep line Sort points in x direction and traverse the points in decreasing X direction. Keep track of the largest y coordinate at every step - call it $Y(i)$ and for point p_{i+1} we test if $y(p_{i+1}) > Y(i)$. It is maximal iff the test succeeds.

The proof of correctness is from definition and the running time is $O(n \log n)$ for sorting and $O(1)$ for every event.

(ii) If there are h maximal points, design an output sensitive algorithm for finding maximal points.

Find the median line (according to the x coordinates) of the point and find the intersection point with the staircase. This can be computed from the maximum y coordinate of R (we don't have to compute the entire R). Now recursively compute the left and right chains. The recurrence for the running time is

$$T(n, h) = T(n_1, h_1) + T(n_2, h_2) + O(n) \text{ where } n_1, n_2 \leq n/2$$

This yields $T(n, h) = O(n \log h)$

2. Given a set S of n points, we want to find the largest empty disk inside the convex hull of S. Design an efficient algorithm for this problem. **(15)**

3. Given a convex polygon $\mathcal{P}$ with n boundary points, find the longest segment that fits inside $\mathcal{P}$. Your algorithm should run in $O(n)$ time.
Hint: A pair of points on the convex hull is an *antipodal* pair if they can support parallel tangents. **(15)**
Let the convex hull have the vertices $p_1, p_2 \ldots p_n$ in an anti clockwise order. For each vertex p_i we consider the supporting lines (tangent) ℓ_i - it can be oriented in a range $[\theta_i, \theta_{i+1}]$ (we will consider the subscripts mod n).

Observation: For $i \neq j$. the ranges are non-overlapping and monotonically increasing for $j > i$.

This is key to the analysis of the algorithm based on rotating callipers that we never retrace.

In addition, we must prove that (i) the maximum length segment is between two corner points.
(ii) The maximum distance pair are antipodal, i.e., they support parallel tangents. Note that two vertices p_j and p_k are antipodal if the the ranges $[\theta_j + \pi, \theta_{j+1} + \pi]$ have a non-empty intersection with $[\theta_k, \theta_{k+1}]$.

Moreover, at least one of the parallel lines of an antpodal pair is supported by a convex hull edge. This makes the algorithm discrete.

4. Consider the following scheme for improving the space bound of range search tree in two dimensions. Instead of storing the points in every level to facilitate the one dimensional range-search in y coordinate, we store them in every k-th level where $k \geq 2$ is some integer parameter. stored in at most $2 \log n$ nodes and we do range search in the y direction in every such node. Because of our new storage scheme, for each canonical interval, we may have to do multiple range search in the y direction. For example if $k = 2$, then for each canonical level we may have to descend one level (if the points are not stored in that level) and do *two* y-range searching instead of *one*. **(10+5)**
(i) Analyse this scheme to derive bounds for space and query time in terms of n and k.
(ii) What is the best query time for linear space ? By storing points at every k-th level, we may have to do the y-range search over a subtree of depth k, i.e. the points have to be picked up from 2^k nodes (in the example $k = 1$). So while the space is $O(\frac{n}{k} \log n)$ instead of $O(n \log n)$ in the original scheme, the search time increases as follows - for canonical intervals in depths $i \cdot k + 1, i \cdot k + 2 \ldots (i+1) \cdot k$ where $0 \leq k \leq \frac{\log n}{k}$, we may have to do $2^k + 2^{k-1} + \ldots 1 = O(2^k)$ y-range searches. Each of them costs $O(\log n)$ time, and so the overall cost for all the depths = $\frac{\log n}{k} \cdot O(2^k \log n)$ which is $O(\frac{2^k}{k} \cdot \log^2 n)$. Note that for $k = 1$ (the original range search scheme) this is $O(\log^2 n)$.

(ii) To keep the space linear, k must be $\Omega(\log n)$ which is $\epsilon \cdot \log n$ for some fixed $\epsilon > 0$. Plugging this into the bound for query time, we obtain $O(2^{\epsilon \log n}/(\epsilon \cdot \log n) \cdot \log^2 n)$ which is $O(n^\epsilon \log n)$. We can actually write this as $O(n^\epsilon)$ since $\log n$ is subsumed by n^ϵ.
In summary, the linear space scheme results in a better than $\sqrt{n}$ bound given by k-d trees but k-d trees store a point *exactly* once.
Note Most common mistake was to take $k = \log n$, (rather than $\Omega(\log n)$ to make it linear space, but that would lead to super-linear search time, which is worse than naive method.